```go
package main

import    (
    "fmt"
    "errors"
    "os"
    "math"
)

type (
    lmnt struct {
          val int
          pos int
    }
    deque []lmnt
)

func initDeque() deque  {
     return make([]lmnt, 0, 0)
}

func (dq *deque) PushLeft (w lmnt) {
    *dq = append([]lmnt{w}, (*dq)...)
}

func (dq *deque) PushRight (w lmnt) {
    *dq = append(*dq, w)
}

func (dq deque) GetLeft() lmnt {
    if dq.Empty()  {
          return lmnt{math.MinInt64, -1}
    } else {
        return dq[0]
    }
}

func (dq deque) GetRight() lmnt {
    if dq.Empty()  {
          return lmnt{math.MinInt64, -1}
    } else {
        return dq[len(dq)-1]
    }
}

func (dq *deque) RemoveLeft () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    }  else  {
        *dq = (*dq)[1:]
        return nil
    }
}

func (dq *deque) RemoveRight () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    } else {
        *dq = (*dq)[:len(*dq)-1]
        return nil
    }
}

func (dq deque) Empty() bool  {
    return len(dq) == 0
}
```

```
67
68  func main() {
69       f, err := os.Open("numbers.dat")
70      if err != nil {
71          return
72      }
73      defer f.Close()
74      var k int
75      _, err = fmt.Fscanf(f, "%d\n", &k)
76      if err != nil  { return }
77      var x int
78      win:= initDeque()
79      for i:= 0; i < k; i++  {
80          _, err := fmt.Fscanf(f, "%d\n", &x)
81          if err !=nil  { break }
82
83          for !win.Empty() &&  win.GetRight().val <= x {
84              win.RemoveRight()
85          }
86          win.PushRight(lmnt{x, i})
87      }
88      fmt.Println(win.GetLeft().val)
89
90      for i:= k; ; i++ {
91          _, err := fmt.Fscanf(f, "%d\n", &x)
92          if err != nil  { break }
93
94          if win.GetLeft().pos == i-k {
95              win.RemoveLeft()
96          }
97
98          for !win.Empty() &&  win.GetRight().val <= x {
99              win.RemoveRight()
100         }
101         win.PushRight(lmnt{x, i})
102
103         fmt.Println(win.GetLeft().val)
104     }
105 }
```

numbers_03.go

```
package main

import    (
    "fmt"
    "errors"
    "os"
    "math"
)

type (
    data struct {
          val int
          pos int
    }
    lmnt struct {
        d data
        l *lmnt
        r *lmnt
     }
    deque struct {
        left *lmnt
        right *lmnt
    }
)

func initDeque() deque  {
     return deque{nil, nil}
}

func (dq *deque) PushLeft (d data) {
     if (*dq).Empty()  {
        (*dq).left = &lmnt{d, nil, nil}
        (*dq).right = (*dq).left
    } else {
        (*(*dq).left).l = &lmnt{d, nil, (*dq).left}
        (*dq).left = (*(*dq).left).l
    }
}

func (dq *deque) PushRight (d data) {
     if (*dq).Empty()  {
        (*dq).left = &lmnt{d, nil, nil}
        (*dq).right = (*dq).left
    } else {
        (*(*dq).right).r = &lmnt{d, (*dq).right, nil}
        (*dq).right = (*(*dq).right).r
    }
}

func (dq deque) GetLeft() data {
    if dq.Empty()  {
          return data{math.MinInt64, -1}
    } else {
        return (*dq.left).d
    }
}

func (dq deque) GetRight() data {
    if dq.Empty()  {
          return data{math.MinInt64, -1}
    } else {
        return (*dq.right).d
    }
}

```

```
func (dq *deque) RemoveLeft () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    }
    if (*dq).left == (*dq).right  {
          (*dq).left, (*dq).right = nil, nil
     } else {
        (*dq).left = (*(*dq).left).r
    }
    return nil
}

func (dq *deque) RemoveRight () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    }
    if (*dq).left == (*dq).right  {
          (*dq).left, (*dq).right = nil, nil
     } else {
        (*dq).right = (*(*dq).right).l
    }
    return nil
}

func (dq deque) Empty() bool  {
    return dq.left==nil
}

func main() {
     f, err := os.Open("numbers.dat")
    if err != nil { return }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    win:= initDeque()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }

        for !win.Empty() &&  win.GetRight().val <= x {
            win.RemoveRight()
        }
        win.PushRight(data{x, i})
    }
    fmt.Println(win.GetLeft().val)

    for i:= k; ; i++ {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }

        if win.GetLeft().pos == i-k {
            win.RemoveLeft()
        }

        for !win.Empty() &&  win.GetRight().val <= x {
            win.RemoveRight()
        }
        win.PushRight(data{x, i})
        fmt.Println(win.GetLeft().val)
    }
}
```

```
package main

import    (
    "fmt"
    "errors"
    "math"
)

type (
    lmnt int
    ring struct {
          row []lmnt
          currentPos int
     }
)

func initRing(len int) ring {
     var r ring
     r.row = make([]lmnt, len, len)
     r.currentPos = 0
     return r
}

func (r ring) Empty() bool  {
    return r.Length() == 0
}

func (r ring) Length() int  {
    return len(r.row)
}

func (r ring) GetCurrent() lmnt {
    if r.currentPos >= r.Length() || r.currentPos < 0 {
          return math.MinInt64
    } else {
        return r.row[r.currentPos]
    }
}

func (r *ring) RemoveCurrent () error {
    if r.currentPos >= r.Length() || r.currentPos < 0 {
        return errors.New("Invalid removing")
    }  else  {
        (*r).row = Delete((*r).row, (*r).currentPos)
        return nil
    }
}

func Delete(s []lmnt, i int) []lmnt {
    return s[:i+copy(s[i:], s[i+1:])]
}

func (r *ring) Forward (step int) error {
    if (*r).Empty() {
        return errors.New("Invalid step")
    }  else  {
        (*r).currentPos = ((*r).currentPos + step) % (*r).Length()
        return nil
    }
}

```

```
62  func main() {
63      var N, M int
64      fmt.Print("Enter N: ")
65      fmt.Scanf("%d\n", &N)
66      fmt.Print("Enter M: ")
67      fmt.Scanf("%d\n", &M)
68
69      r:= initRing(N)
70      for i, _ := range r.row  {
71          r.row[i] = lmnt(i+1)
72      }
73
74      for i:= 0; i < N; i++ {
75            r.Forward(M-1)
76            fmt.Println(r.GetCurrent())
77            r.RemoveCurrent()
78      }
79  }
```

josephus_01.go

```
package main

import    (
    "fmt"
    "errors"
    "math"
)

type (
    lmnt struct {
          n int
          next *lmnt
     }
    ring struct {
          current *lmnt
          len int
     }
)

func initRing(len int) ring {
     return ring {nil, 0}
}

func (r ring) Empty() bool  {
    return r.Length() == 0
}

func (r ring) Length() int  {
    return r.len
}

func (r ring) GetCurrentValue() int {
    if r.current == nil {
          return math.MinInt64
    } else {
        return (*r.current).n
    }
}

func (r *ring) RemoveNext () error {
    if r.Empty() {
        return errors.New("Invalid removing")
    }
    if r.Length() > 1 {
       (*(*r).current).next = ((*(*r).current).next).next
    } else {
    // r.Length() == 1
          (*r).current = nil
    }
    (*r).len--
    return nil
}

func (r *ring) PopNext () (int, error) {
    if r.Empty() {
        return 0, errors.New("Invalid removing")
    }
    val:= (*(*(*r).current).next).n
    return val, (*r).RemoveNext()
}

```

```
62  func (r *ring) InsertFront (value int) {
63      if r.Empty() {
64          (*r).current = new(lmnt)
65          (*(*r).current).n = value
66          (*(*r).current).next = (*r).current
67      } else {
68            p:= &lmnt{value, (*(*r).current).next}
69          (*(*r).current).next = p
70      }
71      (*r).len++
72  }
73
74  func (r *ring) OneStep () error {
75      if (*r).Empty() {
76          return errors.New("Invalid step")
77      }  else  {
78          (*r).current = (*(*r).current).next
79          return nil
80      }
81  }
82
83  func main() {
84      var N, M int
85      fmt.Print("Enter N: ")
86      fmt.Scanf("%d\n", &N)
87      fmt.Print("Enter M: ")
88      fmt.Scanf("%d\n", &M)
89
90      r:= initRing(N)
91      for i:= N; i > 0; i-- {
92          r.InsertFront(i)
93      }
94
95      for i:= 0; i < N; i++ {
96            for j:= 0; j < (M -1)% r.Length(); j++  { r.OneStep() }
97            if tmp, err:= r.PopNext(); err == nil  {fmt.Println(tmp)}
98      }
99  }
```